

## Block 2, Part 2: Client side: JavaScript and the DOM

---

*Peter Thomson (2017), Dave McIntyre (2019) and Stephen Rice (2021, 2022)*

### 1 Introduction

In Part 1 of Block 2, we examined various architectures and components that make up different approaches to web architecture. In this second part of Block 2, we will focus on the client side of this architecture, particularly the most common and familiar client 'the browser'. To demonstrate how this type of client can be enhanced for more functionality, we will start to explore further the client-side programming language 'JavaScript' we encountered in Block 1. The JavaScript language consists of small programs or 'scripts' that can be embedded into HTML web pages. When such a web page is read by a browser, the script is also executed on the client, providing some form of dynamic functionality. The amount of JavaScript on a page can range from none to more complex web sites that are now built entirely from JavaScript that generates all the HTML. In Block 1, you encountered JavaScript utilised to create an image slideshow.

**Note:** JavaScript is not Java. Despite the similarity in names these are two completely different programming languages.

We do not provide a broad investigation into JavaScript here, but rather we will focus on a few selected mainstream and core facilities that typify how JavaScript is used in web applications, particularly on adding processing to the type of HTML forms that you encountered in Block 1.

We expect that you have some previous experience of how a programming language works, using a language such as Sense, Scratch, Python or Java; if you have not, you may find it useful to refer to the 'Introduction to JavaScript' guide, available from Reading and reference resources for Blocks. This guide covers the use of variables, arithmetic expressions, Boolean simple and compound expressions, assignment statements, loops and selection structures, and simple mathematical functions. It also covers how data can be stored in arrays.

There are also a number of guides to JavaScript available through the OU library that you can use for further reading. Please use the Block 2 Forum to recommend any resources that you found useful.

You can also use the forum to discuss the concepts being explained here and to seek or offer help with any problems.

#### **For this part of Block 2, you will need:**

- A suitable plain text editor, such as Brackets or Notepad ++. You can discuss any queries about text editors in the Block 2 Forum.

- An up-to-date version of the web browsers Firefox and Chrome. It is often useful to see the small differences that may exist between the different browsers.
- You will also need to download and save the activity files from the Block 2 Part 2 Resources folder on the module website. Then unzip them into your working folder for Block 2.

## 2 Learning outcomes

On successfully completing this part, you should be able to:

- describe more nuanced levels of thick and thin application clients
- discuss a range of extensions and add-ons that are commonly used by developers with web browsers and explain their purpose
- describe how a client-side language such as JavaScript is embedded into web pages and some of the basic operations it may perform
- discuss how HTML gives structure to a web page
- describe how the 'Document Object Model' provides a hierarchical representation of a web page and identify ways in which this can be used by JavaScript
- write or adapt simple JavaScript scripts to operate on elements within a web page in response to events.

## 3 Clients: The 'thick' and the 'thin'

In Part 1 of Block 2, we explained that clients with less functionality are often described as 'thin' while others can be thought of as 'thick'. This distinction, however, is becoming less clear as some thin clients such as web browsers that handled early HTML standards, have become more and more sophisticated. First, we should compare the main strengths and weaknesses of using thin and thick clients.

### Thin clients

A thin client has little internal logic which means it will not be able to do very much work and will perform only quite simple functions. The thinnest client might only present information in a browser. The project from Block 1 where a web browser simply renders HTML, images, etc., as pages is a good example of this. Every operation on the client side requires an HTTP request to be sent to the server that performs the operation and then returns new information (such as a new web page) to the client. So a thin client typically has frequent exchanges with the server, with the user waiting for the server to respond between each update on the client side.

As the client does very little processing, it is not necessary to have great processing power on the client side. The server, however, is charged with performing a lot of the processing required and must do this for all clients making requests. As the thin client performs little processing, it can also be a relatively simple and small piece of software that will rarely fail.

## Thick client

A thick client may perform a significant amount of processing on the client side and thus puts greater demands on the client side machine. As the thick client performs much of the processing required, the operations carried out on behalf of the user frequently do not require, or wait on, any interaction with the server.

When first invoked a thick client may, depending on the application, need to download an initial set of data from the server on which to operate.

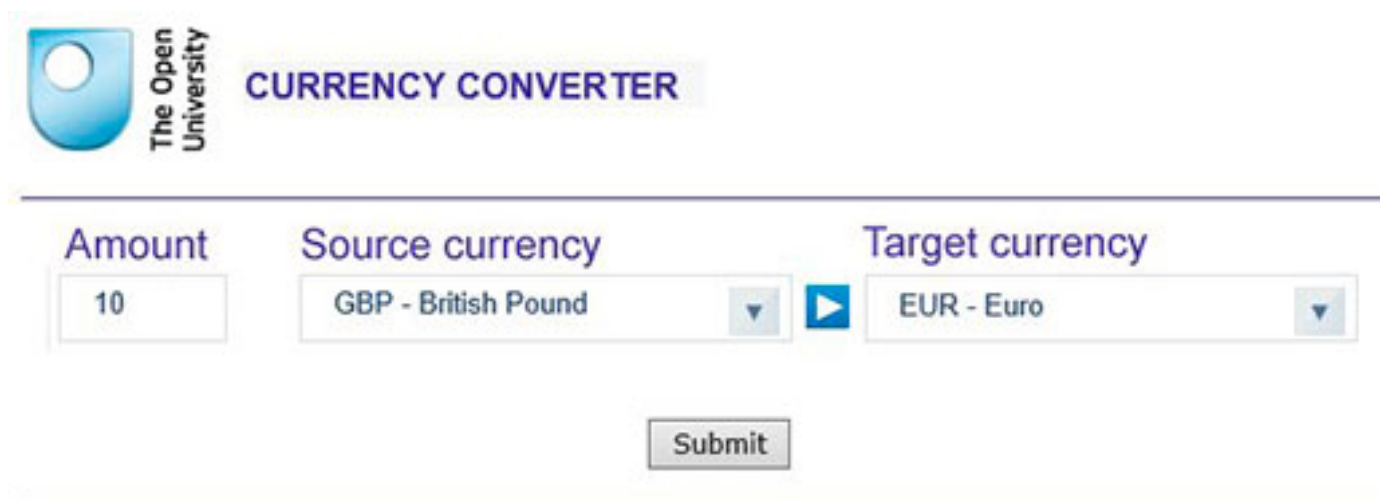
The server will usually validate this request for data and authenticate the user before supplying it from a database, or requesting it in turn from a database server.

This type of approach is used, for example, in interactive applications such as Microsoft's Word Online where the thick client needs to continually process and display changing data.

A dedicated thick client usually performs substantial processing, and would be quite a complex piece of software that could be difficult to install, run and manage across a large number of different client machines and their (usually) different environments. Thicker clients based on a web browser now provide a much simpler way of developing and distributing new software.

## Currency conversion example application

To demonstrate how the operation of an application can change according to the type of client used, consider an online service for currency conversion. The service provides a user interface with a form which allows a user to select a source currency, a target currency and an amount of money to convert from the first currency into an amount in the second currency (Figure 1).



The screenshot shows a web interface for a currency converter. At the top left is the Open University logo. To its right is the title 'CURRENCY CONVERTER' in blue capital letters. Below this is a horizontal form with three main sections: 'Amount', 'Source currency', and 'Target currency'. The 'Amount' section contains a text input field with the value '10'. The 'Source currency' section contains a dropdown menu with 'GBP - British Pound' selected. The 'Target currency' section contains a dropdown menu with 'EUR - Euro' selected. A blue play button icon is positioned between the source and target currency dropdowns. Below these fields is a 'Submit' button. The entire form is enclosed in a light blue border.

Figure 1 Currency converter interface

 Maximise

After the interface displays the currencies selected, the sum to be converted and the result of the conversion, the user then submits the form (Figure 2).

---

Result 10.00 GBP = 11.7893 EUR

---

Figure 2 Currency converter result web page

 Maximise

This type of application only requires a thin client – just a simple web browser to display the page – as the conversion request is handled by the server. The server can also ensure that it is handling up-to-date data.

To see a reasonably up-to-date conversion, try <https://www.google.co.uk/search?q=GBPEUR>.

Now suppose that we want to do more processing of the exchange rate information, such as tracking changes over time, or exploring how currencies are linked to each other. The server can supply us with a thicker client, plus the data, and we can spend the next hour exploring these changes in exchange rates without the server having to do any more processing, or supplying any more data.

Thick and thin clients are simply two ends of a spectrum, with the thinnest clients needing to only present the data supplied to them, usually in HTML format with CSS controlling the presentation. The server supplying the HTML for the thin client handles the data, business and presentation logic and delivers the final result as a single web page, together with the CSS for display. As the client gets thicker, however, it handles more of the presentation logic, perhaps even business logic, and the server handles less. For the thickest clients, the server handles only the data. Everything else is done by the thick client.

## Activity 1 Thin versus thick clients

 Allow 30 minutes for this activity

Consider the following applications or websites that you might run on your own machine and consider how 'thick' or 'thin' you would consider it to be:

- Google Earth
- [www.OpenCycleMap.org](http://www.OpenCycleMap.org)
- Skype
- Facebook
- BBC News

How much do you think each of these depend on the client and how much the server for creating the display and user interaction? Consider how long it takes to download and start the application the first time you use it – a large download and a long set up time usually suggests a thick client, or it might just suggest bad design.

What are the technical strengths and weaknesses (not the content, but how that content is delivered and presented) of each approach? Think about the complexity of the user interaction available.

Discuss these 'clients' in the Block 2 Forum.

## 4 Programming language types

A program is a complete set of instructions for performing a task. When a computer is 'running' or 'executing' a program, the computer is executing the instructions contained in the program.

However, the types of programming language in use today (often called 'high-level languages') cannot be directly understood by a computer's processor. A computer processor, instead, understands machine language, so the instructions in a high-level program language have to be translated or interpreted into machine language in order to be executed.

A simple one line program might be an alert, such as the JavaScript we encountered in Block 1 Part 6:

```
alert("Hello world");
```

This script has to be interpreted by JavaScript into the machine language that the processor can understand.

Before we start any scripting, it is worth quickly running over some of the fundamental concepts behind programming languages and programs.

In general two types of computer languages can be distinguished: **interpreted languages** (such as JavaScript) and **compiled languages** (such as C).

### 4.1 Interpreted languages

Scripts are interpreted at execution time. That is, each time the script is run, the script is converted into code that the processor can understand. There is a wide range of interpreted languages (such as JavaScript, PHP and Python to name just a few). Some are generic and applicable in a wide range of circumstances. For example, JavaScript was originally developed as a script to use inside web pages, but it can now be used as a server side language as well.

Other scripts are tailored to be used for specific purposes. PHP is designed to create dynamic web pages and runs on a web server. The key distinguishing property of interpreted languages is that they are translated from the programming language into processor instructions when the script is executed. This process is termed 'interpretation' of the program.

One result of this need for interpretation is that errors in the code may not become obvious until the script is run.

### 4.2 Compiled languages

Compiled languages such as C, C++ or Java are important in providing the operating systems and very efficient applications in mobile devices.

Compiled languages are translated to an executable file before execution time. The translation from the program into the code that the processor understands happens just once, and then each time the program is run it uses this previously translated or compiled code.

Source code refers to the programming statements written as text by a programmer. The translation process is called 'compilation' and is performed by a software tool called a compiler. The compilation process generates an executable file from the source code of the program. A major advantage of the compiled approach is that the program can be compiled once and run as many times as required without having to translate the high-level programming code into processing instructions each time the program is run. Another advantage is that many errors should be reported at the compile stage.

**Note:** we will not be studying compiled languages further in this module.

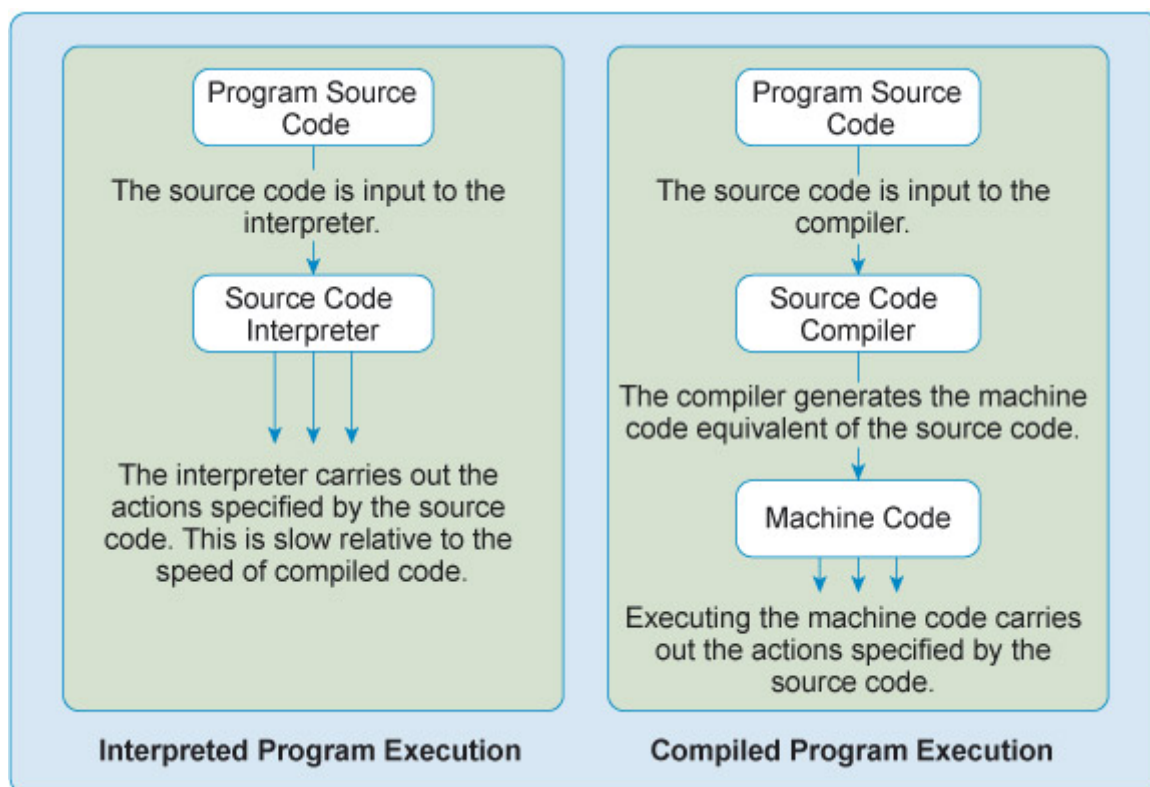


Figure 3 Interpreted and compiled program execution

## 5 Browser-based clients

The web browser is designed to collect all the separate items of information and instructions that go to make up a web page, and then assemble them to produce the web page that is presented to the user.

This is quite a sophisticated application that has evolved from simply displaying information, to becoming one of the main interfaces for computer user interaction.

There are many different web browsers available, some with specialist functions such as enhanced security or parental controls, yet only a few products dominate the market. In 2017, it is reported that more than half of desktop and mobile web users are using Google Chrome. This is almost 3.5 times more users than Safari, which comes second (StatCounter, cited by Statista 2017). A Chinese browser called UC Browser comes third, Firefox fourth, Internet Explorer fifth and Opera sixth.

Therefore, in terms of market share, it is very wise to check that anything designed for the web will run on Google Chrome!

Chrome also has another advantage for us as we study the Web and web applications – it implements many of the most recent developments in HTML and CSS.

The working groups of people that develop these browsers share a lot of code, so there are many similarities between some browsers. Chrome, Edge and Opera share the same HTML engine (Blink). Safari and UC Browser share the same HTML engine (WebKit). Blink and WebKit are also very closely related to each other.

In Block 1 you created some static web pages using HTML and CSS and also looked at forms, which are used to submit information to a server. Both HTML and CSS are very frequently used in combination with JavaScript, a client side programming language that allows much greater functionality and interactivity to be supported within web pages on the client.

JavaScript will work in most modern browsers. For this block, we will be using the current versions of Chrome and Firefox, but do try the examples in other browsers if you have them.

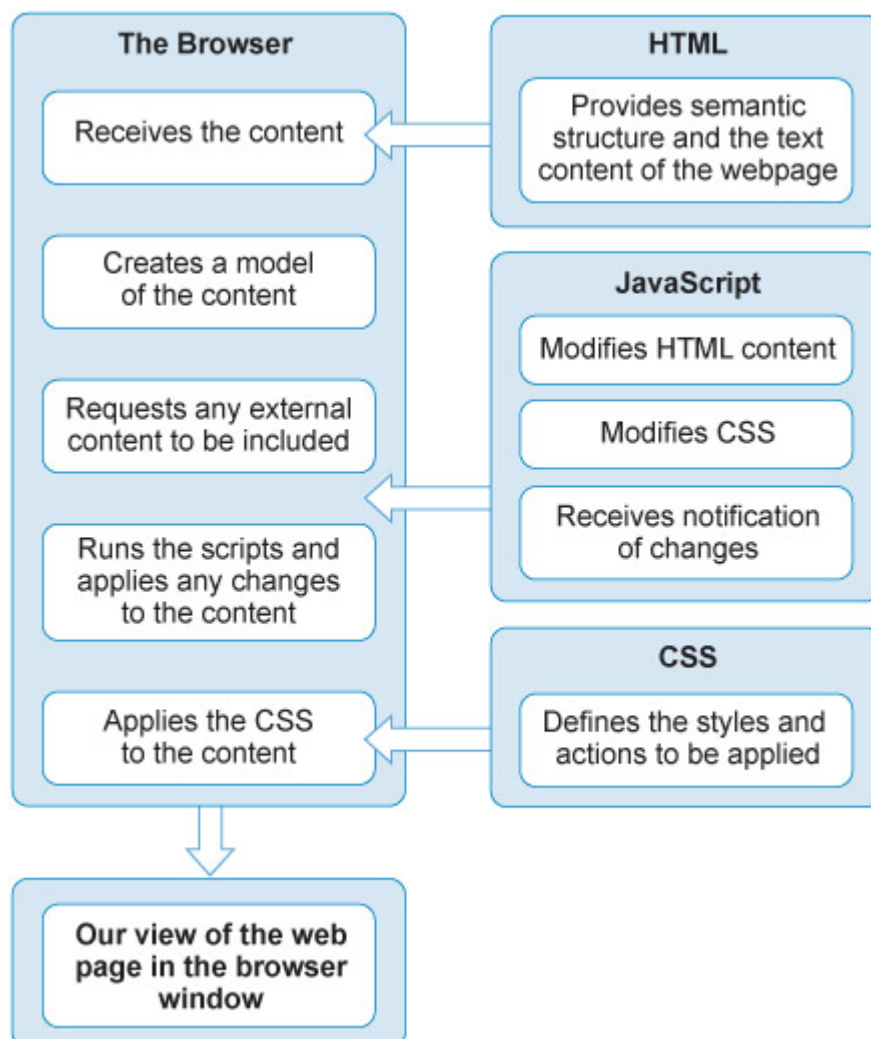


Figure 4 The browser utilises the HTML, JavaScript and CSS information

Most browsers also support the use of plug-ins, addons or extensions that provide some additional functionality, either in the presentation of web content or in the management of it by a user. New extensions appear regularly. Some work on all browsers, but some are tailored only for a specific browser, such as Firefox or Chrome. Common types of extensions are:

- Ad blockers: these block advertisements from appearing within web pages. One example is Adblock.
- Reference management tools: these help authors to collect and manage lists of reference materials for research. Examples are Zotero and Endnote.
- Bookmarking and curation: these allow users to collect and often annotate web content, either whole pages or elements within pages. Examples are iCloud Bookmarks, Storify and Diigo.
- Web development tools: these offer help to web developers in authoring and monitoring of web content. These can be particularly useful as an aid to debugging and testing HTML, JavaScript and CSS. Increasingly, features first implemented as extensions have become available as developer tools built-in to Chrome and Firefox, so it is worth exploring these first before installing extensions.

## 6 HTML and the Document Object Model

To manipulate the HTML elements that form the current web page, we must first understand something of the way in which the page is represented in the web browser. This representation is known as the Document Object Model (DOM) and is defined by WHATWG and W3C standards that specify how software can access, read and modify any part of the document, as well as the properties of each part.

To build the DOM the web browser first reads the HTML file and its tags into a tree of HTML elements. The distinction between tags and elements is subtle but important: for example, `<b>` is a HTML tag, but `<b>Foo</b>` is a HTML element, built from an opening and closing tag with a text value between the two.

In Block 1, we introduced the ‘well-formed syntax’ of HTML using opening and closing tags to describe HTML elements. Where closing tags are available, these should always be used even though some elements will be displayed correctly without the closing tag. HTML has rules about how to interpret missing closing tags, so missing them out may not always result in differences in page appearance between modern browsers, but could well indicate a mistake in your HTML and the page may not appear how you want.

Not all elements have a closing tag. For example, a meta tag within the head, or an image.

```
<meta charset="UTF-8">
```

```

```

If you are working with XML compliant HTML, then elements like these should be closed thus:

```
<meta charset="UTF-8" />
```

```

```



Any well-formed, that is, correctly nested HTML tag structure like this can be described as a tree structure, and this is shown in Figure 5. Note that this diagram only shows the opening tags for each element.

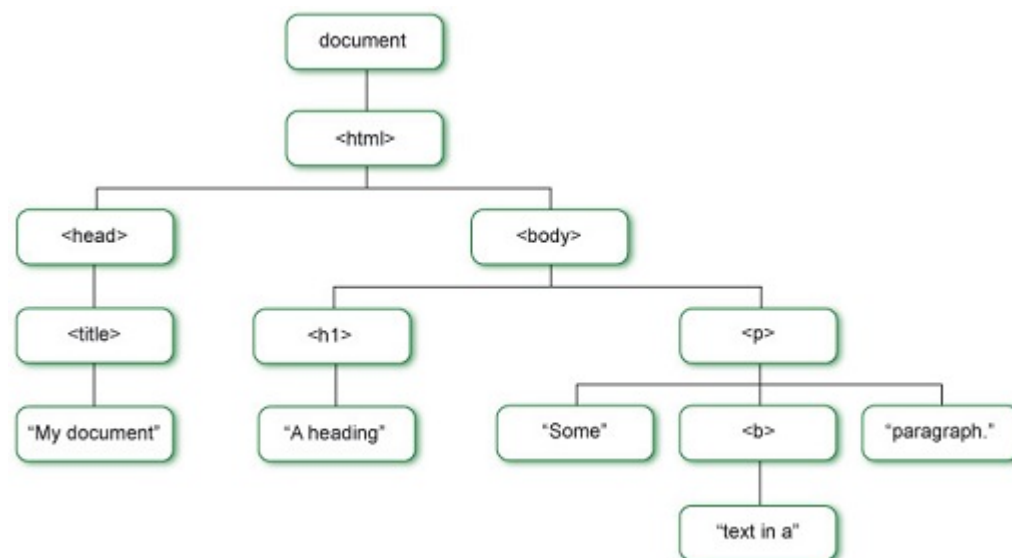


Figure 5 A tree view of the HTML

In a tree representation, each element is known as a node and represents part of the structure of the document. The root of the tree, the document node, is the parent of the HTML node. The HTML node is the child of the document node and is itself the parent of both head and body nodes. The head node is the first child of the HTML node and a sibling of the body node.

An element, plus all those other elements nested within it, is known as an object. An object could also be described as a node plus all its children or descendants, so the document object can also be considered to describe the whole of a HTML document.

Access to the DOM only works reliably if the HTML syntax is all correct, otherwise there will be errors in the tree. Hence the need for regularly validating your code using <https://validator.w3.org/>.

Another way of looking at the tree structure is to extend the concept to the containing window. In this case, the Browser Object Model or BOM has the 'window' property as the root of the tree with the document as a child node, and other properties of the window become available for software to access, read and in some cases modify.

The following diagram represents the whole of the BOM. The document on the second row and its children in the third row represent the DOM, as previously illustrated in Figure 5.

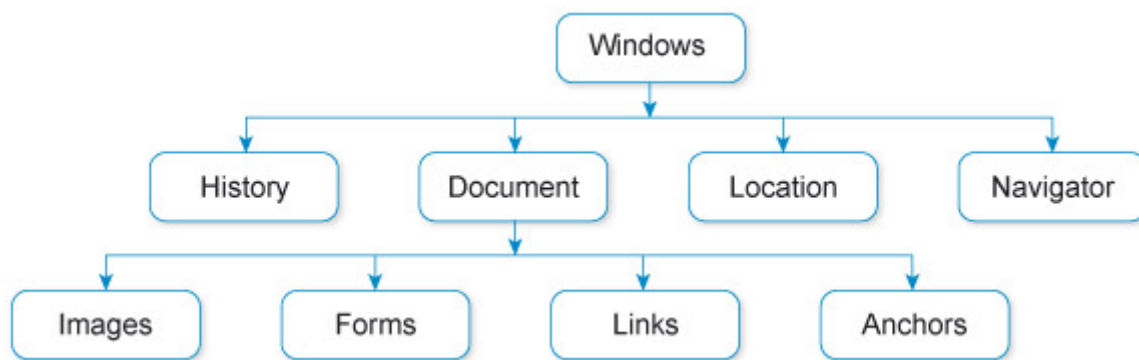


Figure 6 A BOM view of the window

These object models allow any software to manipulate and utilise all the information within a browser window, and to help create dynamic web pages and apps.

Each object within the window, and properties of that object, can be accessed by using dot notation, for example, `window.document.images`. The dot notation traces the path from the root node of the window to the next node that is directly connected to it, down the branches of the tree:

`window.history` (e.g. `window.history.length`) – returns the number of URLs in the history list.

`window.document` (e.g. `window.document.characterSet`) – returns the character encoding of this page.

`window.document.body` – gives access to the body of the document, i.e. the `<body>` tag and its children.

`window.document.forms` – gives access to all the forms in a document.

`window.document.images` – gives access to all the images in a document.

`window.location` (e.g. `window.location.pathname`) – returns the path of the current page.

`window.navigator` (e.g. `window.navigator.appVersion`) – retrieves the browser version (**note**: the navigator object does not relate to navigation; it is named after Netscape Navigator, a predecessor of Firefox).

In many cases, a shortened version is normally used: `window.` is omitted, and the software would use `history.length` or `navigator.appVersion`

It is this dot notation that is used to change and set properties within the window or document. For example, `document.body.style.backgroundColor = "blue"` would change the existing background colour of the whole page to blue.

Using this notation, new elements can be created in the web browser and added to the DOM.

The following is how we use this dot notation to create and add an element to the current page:

```
var buttonElement = document.createElement("button");
var textElement = document.createTextNode("My new TT284 button");
// Set the text of the button by nesting the text element inside it
// i.e. <button>My new TT284 button</button>
buttonElement.appendChild(textElement);
// Add the button to the end of the document body
// i.e. before the </body> tag
document.body.appendChild(buttonElement);
```

Where HTML elements are created by reading HTML tags rather than using `document.createElement`, they can be given a unique 'id' attribute and identified by the value of this attribute. This can be used to look up the element in DOM using `document.getElementById` to obtain its properties or to make changes to them. It is very important, therefore, that each ID is unique within a given document.

In a HTML form, we can start with the ID of an input element to retrieve a value entered by the user, for example:

```
var inputElement = document.getElementById('msquared')
```

which identifies a particular form element with the ID 'msquared'.

Once we have a reference to a HTML element using `document.createElement` or `document.getElementById` or as the target of an event such as a mouse click or change of value, we can use the dot notation to write or read the 'value' property of that element:

```
inputElement.value = '12';

alert(inputElement.value);
```

We can use that to assign the value to a variable:

```
var area = inputElement.value;
```

Creating elements and text nodes can require a lot of code, but the 'innerHTML' property provides a convenient shortcut to allow us add fragments of HTML to the page. Given a reference to a HTML element like a heading, paragraph or span, we can add new content using the dot notation to assign a new value to its

'innerHTML' property, for example:

```
document.getElementById("tt284").innerHTML = "<h2>Hello World</h2>"
```

This retrieves the element identified by the ID "tt284" and then sets the value of the 'innerHTML' property to a string representing a HTML heading (h2) and some text (Hello World). This is a useful shorthand: the browser will create an h2 element and text node for us and replace the existing children of the "tt284" element with this new element. There are two drawbacks to this approach: firstly that screen readers may not be aware of the new content, and secondly that care must be taken when adding untrusted data to the page in case it contains malicious content.

We return later to the concept of the DOM and how we can use it to respond to events, identify elements and manipulate them.

**Note:** When coding, if you use View Source in a web browser it will show you your code. If you Inspect in Developer Tools it will show you an HTML representation of the current Document Object Model – that is, the HTML will be valid and nested, but it may not be the same as your original code. Missing information may be filled in and invalid information removed. Don't rely on just one method to inspect code.

## 7 JavaScript

As described earlier, JavaScript, the most common client-side web technology is an 'interpreted' programming language, often referred to as a scripting language. Below we outline some of the key points behind its dominance in this role:

- JavaScript can be embedded in any HTML page.
- JavaScript can be natively interpreted by all modern browsers, without the use of any extensions or plug-ins.
- Computations and actions can be executed on the client side, and there are situations in which this is preferred, for example, to provide immediate responsiveness in a web page.
- JavaScript is 'mainstream', meaning that it is recognised as a useful language with many practitioners. It is recognised as an evolving standard known as ECMAScript and as having some longevity.

### 7.1 Inserting JavaScript into web pages

There are three common ways of inserting JavaScript code in a web page:

1. Inside an HTML script element:

```
<script>

alert("JavaScript is working in your browser");

</script>
```

2. Inside a JavaScript file, linked from the HTML page that contains a declaration to include the JavaScript file. The 'path/to/file.js' value in this example is the path from the HTML file on the server to the JavaScript file on the same server – note the opening and closing script tag with no text between the two:

```
<script src="path/to/file.js"></script>
```

The JavaScript file will include just the JavaScript code. No HTML is required or allowed.


3. As a value of event and hyperlink attributes in the HTML page:

```
<button onclick="alertFunction()">Test it</button>
<script>
function alertFunction() {
    alert("Example alert box!");
}
</script>
```

```
<a href="javascript:alert('JavaScript is working in your browser');">link</a>
```

JavaScript code in a page is parsed as the page is loaded, just as HTML is parsed. The code may be executed at the time it is loaded into a browser, or it may be parsed and made available to be executed in response to an event. JavaScript provides a wide range of events including clicks, keypresses and mouse movement, input changes and form submission.

## Activity 2 JavaScript 'Hello world'

 Allow 20 minutes to complete this activity

Use your text editor, to open the file helloworld.html that you downloaded from the Block 2 Part 2 Resources folder. Also open the same file in Google Chrome.


Try to match the alerts, console logs and the text that you see on the page, to the HTML code and JavaScript code. Then use developer tools in your browser to explore the HTML representation of the Document Object Model and compare that to the original HTML file. Try to match the additional elements with the JavaScript that added them.

The JavaScript code is all embedded in the HTML, which is rarely what we want. As soon as we become familiar with using JavaScript, the code should be moved to a JavaScript file.

### Important syntax notes:

- The characters “ ” and " are not the same. As with HTML, take care when pasting to and from a word processor as it may change these characters.
- An opening " must be matched by a closing "
- An opening ' must be matched by a closing '
- JavaScript uses different brackets for different purposes. Make sure you use the correct ones. Opening brackets, will always be matched by closing brackets, for example, ({ [ ] })
- Spelling of code words and variables must always be accurate and use the correct upper or lower case characters.
- Punctuation must be 100% correct. There is no room for error.
- If you encounter an error, don't add more code until you have solved that error.
- If you have an error in your code, sometimes web browsers will not show you an error message until you open developer tools and look at the JavaScript console; instead the code will appear to do nothing.
- You can use developer tools in your web browser to see exactly where and when errors occur and even step through your code line by line.
- When writing code, add one line, test, correct, and test again.

### Activity 3 Experiment with this helloworld.html page of HTML and JavaScript

 Allow 20 minutes to complete this activity

In this activity, you are encouraged to experiment with the code provided. A number of suggestions follow. You may find your own.

- Change the messages in the alerts.
- Change the text that is added to the page by the script.
- Move the HTML element within which text is added to the page by the script.
- Open developer tools and locate the JavaScript console. Check that you can see the message that you wrote using `console.log()`.

## 7.2 Where to place JavaScript: HTML, head, body or as a JavaScript file

For JavaScript to work, it first needs to be loaded into the HTML document by the browser before you can use it. If all the HTML must be rendered first, then any JavaScript can be placed at the end of the HTML, just before the closing `</body>` tag. This speeds up the process of displaying the page to the user.

Conversely, if the HTML page needs the JavaScript to create the display, then it makes sense to place the JavaScript earlier in the HTML in the <head> tag of the code.

If the JavaScript is part of a library of code, or used on more than one page, then it should be in a JavaScript (.js) file. If the file is on the same folder or directory as your page you can use a relative URL as show below:


```
<script src="helloworld2.js">
```

But if the file is loaded from another website you will need to use the absolute URL:

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
```


JavaScript code is normally kept in JavaScript files rather than contained in a <script> tag. You never want to see the same JavaScript code repeated in multiple places as that leads to issues with the code. If, however, a script is only required for single use on a single page it may be placed inside a <script> tag on that page.

## Activity 4 Compare what you see on-screen, in View Source and through Developer Tools

 Allow 10 minutes to complete this activity

Use your text editor to open the file **helloworld2.html** from the Block 2 Part 2 Resources folder. Also open the same file in your browser. You will need to compare what you see on screen with what is displayed by View Source, and what is displayed by Developer Tools when you inspect an element. Also open the files **helloworld2.js** (helloworld2.html).

## Activity 5 Experimenting with JavaScript functions and events

 Allow 15 minutes to complete this activity

**Copy both helloworld2.html and helloworld2.js to a new folder before you start working on them, then load both from the new folder into your text editor. Save the files after editing them and view the changes in your browser.**

- Change the IDs of the elements.
- Change the names of the functions.
- Change the messages produced by the functions in the HTML file and those in the JavaScript file.
- Using View Source will show the original source with empty elements. Using Inspect from the Developer Tools will show HTML reflecting the current DOM.
- As you work, check the JavaScript console in Developer Tools for any error messages.

You need to be sure that you can make all these changes without introducing any errors, so that everything still works.

If you do get errors, check that your spellings in the document and in the script match each other. Make sure that you haven't changed any of the punctuation.

## 7.3 Using JavaScript in HTML

We will be using an example of an imaginary flooring supplies company 'Flooring 4-u R-us', which has a website for potential customers

### Flooring 4-u R-us



#### Engineered oak wooden flooring

Pack size 1.36m<sup>2</sup>

Pack cost £ 20.10

Area required =  m<sup>2</sup>

Submit

Figure 7 Web page for costing wooden flooring

 Maximise

The form in Figure 7 is intended to be used by a potential customer ('the user') to state the area in square metres of flooring they require. This web page is for a specific type of 'Engineered oak wooden flooring' supplied in packs, each of which covers 1.36 m<sup>2</sup> of floor area. The page would probably be part of the supplier's catalogue of different floorings. This form is intended to give the user a cost for an amount of oak flooring they want to order.

On this web page, the user types a number representing the floor area in metres squared, let's say '5'; the user clicks 'Submit', and then waits for the server to respond. The response might look like that shown in Figure 8. You can see that the application has calculated the number of packs of flooring required and has added VAT and a delivery charge to arrive at a total cost.



# Flooring 4-u R-us



## Engineered oak wooden flooring

Pack size 1.36m<sup>2</sup>

Pack cost £ 20.10

### Your order cost

Packs required = 4

Cost of packs = £ 80.40

Value added tax = £ 16.08

Delivery charge = £ 35.00

Total = £ 131.48

[Submit order](#)

Figure 8 Response to request for costing

[Maximise](#)

This information will be useful to the customer, but if the server performs the calculation, then every time the customer wants to see the cost of any type of flooring a request has to be sent to the server and the customer has to wait for the response to be returned and displayed. Perhaps even worse is that if the customer changes the area required, then a new request has to be sent to the server to recalculate a new cost.

The web page in Figure 8 illustrates a basic need for a simple local 'behaviour'. You might reasonably expect that the information displayed in the web page in Figure 8 is put together by the web server, which gets the image of the particular flooring, the pack size and pack cost and puts these together with a standard heading. We will investigate exactly how a server might do that later in this block, but for now the key point is that all the information required to calculate the price can be made available as part of the initial web page. If this is the case, then when the user types in the area of flooring required, the price can be calculated by the browser on the client side without the need to send any request back to the server. That is exactly the type of 'local behaviour' JavaScript supports within a browser.

## 7.4 Adding JavaScript to a form

In this next example, instead of relying on the server to complete the calculations and return the result to the browser each time as a request and response sequence, we will use JavaScript to do the calculations using data supplied by the server.

You will need access to the file **flooring.html** from the Block 2 Part 2 Resources. This depends on files named **flooring.css**, **flooring.png** and **flooring.jpg** to appear as intended.

All the JavaScript code in this script is written as a single function called 'calculateOrder'. This starts out by defining some constant values for delivery charge, the cost of a pack of this flooring, the rate of UK value-added tax (VAT) as a rational number, and the area in square metres that a single pack of this flooring covers. These values could be included in the web page served by the server that might retrieve the values from a database.

```
var packArea = 1.36;  
var packCost = 20.1;  
var vatRate = 0.2;  
var delivery = 35;
```

The area of flooring that the user has typed into the form has to be retrieved from the form and converted to a number. This is done by using the ID of that input element in the form and then using the dot notation to identify the element value:

```
var inputElement = document.getElementById("msquared");  
  
var area = Number(inputElement.value);
```

We have used a JavaScript function 'Number' to convert the value property (a string value) into a number. This function (see 'JavaScript Number Object') actually creates a 'number object' based on the numeric value passed to it as an argument. If the value cannot be understood as a number then the value 'Not-a-Number' ('NaN') is returned. So for safety we should also be testing for this.

Having got the area of flooring required out of the form, then the number of packs of flooring is then calculated:

```
var packs = Math.ceil(area / packArea);
```

which divides the floor area by the area covered by one pack and then uses the JavaScript 'Math.ceil' method to round the number upward to the nearest integer (or in this case pack number). The resulting integer is then put into the 'packs' form field on the next line. The flooring cost, VAT and grand total are then calculated.

These totals, together with the fixed delivery charge, are then put into the corresponding 'span' elements. When this is done, the JavaScript method 'toFixed()' is called to restrict the number of decimal places presented to two, and converts into a string so that we can display the number.

So, overall, the `calculateOrder()` function, when called in response to a 'click' event, will get the flooring area required from the form, calculate all the costs based on this area and will then put the results ('cost', 'VAT', 'delivery' and the final 'total' cost) back into the appropriate span elements that follow the form. These elements are identified by their id, using `document.getElementById` ('the target id') and then we assign the 'innerText' property of this object its new text value. This works similarly to the 'innerHTML' property but is appropriate when we don't want to insert HTML elements, only to change text in the page.

The function is parsed and checked for syntactic correctness when it is loaded into the browser, but it is not executed until the function is invoked by some other subsequent action such as an onclick event.

In the introductory section, we used a link to generate a 'click' event. There are different ways this can be done but here we have added a button labelled 'Calculate order' to the web page. When this button is clicked the 'calculateOrder' function is called. The 'onclick' attribute of a button contains code which is executed when the user clicks on the button. This allows code to be called whenever the button is clicked, rather than when the web page is loaded.

## Activity 6 Flooring cost calculation in JavaScript: adding validation to the form

 Allow 30 minutes for this activity

For this activity, you will need access to the file `flooring.html` in your downloads from the Block 2 Part 2 Resources folder. In your browser, load the form and test it making sure that you understand how it works. Also refer back to how we created an alert earlier in this section.

As previously mentioned, the value of 'area' should be checked after it is converted to a number to make sure it is a real number. Alter the form so that the value returned by the 'Number' function is checked to ensure it is a number. If the value is NaN (not a number) then an Alert can be used to warn the user the value has to be a number and, of course, the calculations can't be made if it is not a number.

Hint: here is an outline of the appropriate code:

```
var area = Number(inputElement.value);
if (isNaN(area)) {
    // some code to alert the user of the problem
} else {
    // continue with calculation and report
}
```

Try and modify the code yourself before checking with the solution. Take care over the capitalisation of 'isNaN'.

You can discuss your solution in the Block 2 Forum.

## 7.5 Some DOM methods for accessing HTML elements

We will now look quickly at some useful DOM methods for accessing elements in the current document.

---

<code>getElementById()</code>	Accesses the first element with the specified id
<code>getElementsByName()</code>	Accesses all child elements with a specified name
<code>getElementsByTagName()</code>	Accesses all child elements with a specified tagname
<code>querySelector()</code>	Accesses the first child element matching a CSS selector
<code>querySelectorAll()</code>	Accesses all child elements matching a CSS selector

---

As you have already seen, the first of these DOM methods searches the document for the first HTML element with the given value of 'id' attribute. Note that this function is case-sensitive, although we talk about IDs, the method is called `getElementById` **not** `getElementByID`.

To get the value entered into the metres squared field, we used:

```
document.getElementById('msquared').value;
```

In the HTML form the field is specified by:

```
<input type="text" name="msquared" id="msquared" size="3" maxlength="4" />
```

So, we obtain a reference to a HTML element and build up a path through that element's properties by joining the name of each one together separated by a dot, in this case adding `.value` to read the value of the input element.

IDs are used to refer to the element within the document from JavaScript and must be unique to the document. If the ID used was not unique, then the first element with the specified ID would be obtained, and that would make it difficult to access subsequent elements.

Of the other functions, `getElementsByName`, `getElementsByTagName` and `querySelectorAll` match more than one element and return results as an array. Note that these begin 'getElements' (plural) not 'getElement' (singular).

The `getElementsByName` function is useful for HTML inputs as they have a 'name' attribute which is used when sending data to a web server, which may be non-unique (which as we saw in Block 1 Part 6 allows for checkbox groups, radio button groups, etc.).

Unlike `getElementById`, the other functions are available on the 'document' object but are also available on any other HTML element and search the child nodes of that element.

The `querySelector` functions are more flexible as they allow use of CSS selectors such as 'span', '#id' or '.class' to do everything the other functions can do and more.

Given a target HTML element (a button in the form), the JavaScript below gets all the sibling elements that are of HTML tag type 'span'. Sibling elements are those with the same parent, so it does this by accessing the parent node of the target (the form element), and then querying all elements matching the CSS selector 'span', that is, all elements of tag name 'span' that are nested within the form element. It then uses a loop to write each of these elements to the JavaScript console where the element and its properties may be explored further.

This is a rather convoluted example to illustrate a point – you may be able to find a simpler solution using other DOM functions.

```
<script>
function logSiblingSpanElements(target) {
    console.log("Sibling span elements:");
    var elements = target.parentNode.querySelectorAll("span");
    for (var element of elements) {
        console.log(element);
    }
}
</script>
```

Add a button to the HTML to call this script (with type button to prevent it submitting the form):

```
<button type="button" onclick="logSiblingSpanElements(event.target)">
    Log sibling span elements to console
</button>
```

You will need to open the JavaScript console in your web browser developer tools to view what it produces.

## Activity 7 Listing a form's 'input' elements

 Allow 60 minutes for this activity

Place the button just before the end of the form element, and this JavaScript code just before the closing `</body>` tag in the document `flooring.html` from earlier.

Open in your browser and click the button to see what is written to the JavaScript console by the code.

Try and modify the code yourself before checking with the solution. Then try changing the code to search for different elements using different functions and patterns.

## 8 Further reading

There are a huge range of DOM objects and properties. A useful reference source is the Document Object Model page of the MDN (Mozilla Developer Network) website and the many pages linked from there. The w3schools website also provides many JavaScript examples that you can try out online.

You need to be aware that resources such as MDN exist, and are available as a source of reference, but you don't need any more from them for the moment.

### Activity 8 Using online references

 Allow 30 minutes for this activity

The six parts of Block 2 cover all of the objects, methods and events that you need to complete this block and the assessments in TMAs and the EMA, but you should also make yourself aware of the reference guides available on the web that provide complete and comprehensive coverage. You may find these guides provide you with alternative methods and descriptions that will help you to have a wider view if you have time. Check the block forums and the OU Library to see what up-to-date books are being recommended for further study.

## 9 Summary

This part of Block 2 has focused on the client side of web application architecture from Part 1 of the block. We have experimented with some client side scripting in JavaScript to illustrate how this scripting language can be used to retrieve information from web page elements such as the values in the elements of a form, and also to manipulate the content and DOM structure of a web page in a variety of ways. Most importantly, we have examined aspects of the DOM and how this can be used in JavaScript with forms.

## References

- Mozilla developer network (2017) *Document*. Available at: <https://developer.mozilla.org/en-US/docs/Web/API/document> (Accessed: 3 June 2023).
- StatCounter. (2017) 'Global market share held by leading internet browsers from January 2012 to May 2023'. Available at: <https://www-statista-com.libezproxy.open.ac.uk/statistics/268254/market-share-of-internet-browsers-worldwide-since-2009/> (Accessed: 3 June 2023 ).
- Standard ECMA-262 ECMAScript® (2017) '*Language Specification*, 8th edition' (June 2017). Available at: <https://www.ecma-international.org/publications/standards/Ecma-262.htm> (Accessed: 5 June 2023).

## Where next?

In the next part of Block 2 we will investigate how JavaScript can be used to create useful behaviours in HTML forms. A key improvement to usability will be to 'validate' the data entered into the fields of a form and to provide useful feedback to the user as they complete the form.

---